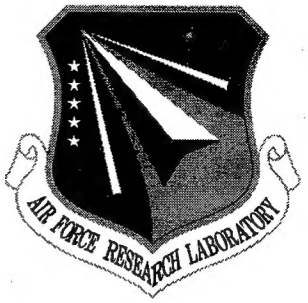
AFRL-IF-RS-TR-1999-102
Final Technical Report
May 1999



CHANGES, CONSISTENCY AND CONFIGURATION IN HETEROGENEOUS, DISTRIBUTED SYSTEMS

Stanford University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. C392

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-102 has been reviewed and is approved for publication.

APPROVED: Craig S. Anken

CRAIG S. ANKEN
Project Engineer

FOR THE DIRECTOR: Northrup Fowler

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Rd, Rome, NY 13441-4114. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

CHANGES, CONSISTENCY AND CONFIGURATION IN HETEROGENEOUS, DISTRIBUTED SYSTEMS

Jennifer Widom

Contractor: Stanford University
Contract Number: F30602-95-C-0119
Effective Date of Contract: 19 May 1995
Contract Expiration Date: 30 November 1998
Short Title of Work: Changes, Consistency and Configuration in Heterogeneous, Distributed Systems
Period of Work Covered: May 95 - Nov 98

Principal Investigator: Jennifer Widom
Phone: (415) 723-7690
AFRL Project Engineer: Craig S. Anken
Phone: (315) 330-4833

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Craig S. Anken, AFRL/IFTB, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 99	Final May 95 - Nov 98

4. TITLE AND SUBTITLE
CHANGES, CONSISTENCY AND CONFIGURATION IN HETEROGENEOUS, DISTRIBUTED SYSTEMS

5. FUNDING NUMBERS
C - F30602-95-C-0119
PE - 62301E
PR - C392
TA - 00
WU - 01

6. AUTHOR(S)
Jennifer Widom

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Sponsored Projects Office
Stanford University
Stanford, CA 94305-4125

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/IFTB
525 Brooks Rd
Rome, NY 13441-4114

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-1999-102

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Craig Anken, IFTB, 315-330-4833

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The objective of the effort was to develop a unifying framework for change management and automatic change notification suitable for diverse heterogeneous information sources, and to demonstrate that the framework can be realized in a flexible, efficient, and scalable manner. Our results are as follows: 1) We designed and implemented a model for change management in heterogeneous information sources. The model, called DOEM (for Delta-OEM) is based on a simple semistructured object model called OEM developed within the companion DARPA-funded Tsimmis project. 2) We developed sophisticated algorithms for detecting changes in semistructured data. Historical information produced by our change detection algorithms is stored in our Lore system (Lightweight Object Repository). Lore is a DBMS suitable for DOEM and OEM data, and was developed in part within the C3 project. 3) We designed and implemented a query language, called Chorel (for Change-Lorel), which is based on Lorel, the query language of Lore. 4) Using our DOEM model, change detection algorithms, and Chorel query language as basic building blocks, we designed and implemented a Query Subscription Service (QSS).

14. SUBJECT TERMS
Distributed Information, Data Integration, Data Warehousing

15. NUMBER OF PAGES
44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

FINAL REPORT

C^3: Changes, Consistency, and Configurations in Heterogeneous Distributed Information Systems

Stanford University

Principal Investigator: Prof. Jennifer Widom

Contents

1 Project Objective

The objective of the effort was to develop a unifying framework for change management and automatic change notification suitable for diverse heterogeneous information sources, and to demonstrate that the framework can be realized in a flexible, efficient, and scalable manner.

2 Results Summary

- We designed and implemented a model for change management in heterogeneous information sources. The model, called *DOEM* (for *Delta-OEM*), is based on a simple semistructured object model called *OEM* developed within the companion DARPA-funded *Tsimmis* project. We leveraged wrapper and mediator technology from the Tsimmis project so that diverse information sources can encapsulate their changes in our DOEM model.

- In order to exploit Tsimmis wrapper and mediator technology we developed sophisticated algorithms for detecting changes in semistructured data. Historical information produced by our change detection algorithms is stored in our *Lore* system (*Lightweight Object Repository*). Lore is a DBMS suitable for DOEM and OEM data, and was developed in part within the C^3 project.

- We designed and implemented a query language, called *Chorel* (for *Change-Lorel*), which is based on *Lorel*, the query language of Lore. Chorel allows users to easily and efficiently query over changes together with data.

- Using our DOEM model, change detection algorithms, and Chorel query language as basic building blocks, we designed and implemented a *Query Subscription Service* (*QSS*), which allows users to subscribe to changes of interest in heterogeneous, distributed information sources, and be notified automatically when the changes of interest occur. QSS provides a number of useful parameters for flexibility in subscriptions.

Sections 4–6 of this report provide accessible but cursory coverage of the results of the project. These sections include motivation and background material, an overview of the C^3 system architecture and its individual components, and a demonstration "walk-through". Section 7 then provides considerable additional details, including theoretical and algorithmic results of the effort as well as further discussion of the implementation of the C^3 prototype.

3 Web Pages and Publications

The project Web page is located at the URL:

`http://www-db.stanford.edu/c3/c3.html`

A selected set of the most relevant publications partially or entirely funded by this project is listed below. All publications also are available by navigating from the Stanford Database Group's WWW home page: `http://www-db.stanford.edu`. Please note that excerpts of publication 4 appear in Section 7 of this report.

1. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montreal, Canada, June 1996.

2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88. April 1997.

3. J. McHugh. S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66. September 1997.

4. S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. *Proceedings of the Fourteenth International Conference on Data Engineering*. pages 4–13, Orlando, Florida, February 1998.

4 Background and Motivation

The vast amount of information available on the World-Wide Web has sparked great interest in the subject of storing and querying heterogeneous and semistructured data. *Heterogeneous* data is characterized by a high degree of autonomy, an absence of a common data model, an absence of standard database control facilities (such as transactions), and differing modes of access. Heterogeneous and other Web-accessible data frequently is *semistructured*, meaning that the data may be irregular or incomplete. In addition to offering access to large amounts of heterogeneous and semistructured information, the Web allows this information to change at any time and in any way. These rapid and often unpredictable changes to the information create a new problem: one of detecting, representing, and querying these changes.

As an example, consider a Web site offering weather information. Recent work on integration of semistructured data allows us to construct a *wrapper* that presents the information at this site using a uniform graph-structured data model called *OEM*, and allows the data to be queried through the wrapper. For example, we can ask "find cities in the Bay Area that had more than half an inch of rain overnight." However, in general people are more interested in weather trends than in past weather data. If the overnight rainfall was half an inch yesterday but was three inches the day before, then chances are this storm is tapering off. However, if there was half an inch yesterday but no rain the day before, this storm may be building. So, the query we would like to ask is "find cities in which the overnight

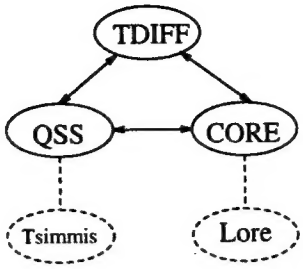


Figure 1: Components of our change management system

rainfall has increased by half an inch over the past two days." This query requires access to previous states of the data, a feature that is not supported by most current semistructured and heterogeneous data management systems, nor by the Web itself. Similarly, perhaps we would like to be notified automatically every time the overnight snowfall in the Sierras has been at least six inches for three nights in a row. Such a "trigger" requires not only access to previous states of the data, but also a method to detect when the trigger should fire.

In the C^3 project at Stanford we have addressed the problem of change management in heterogeneous semistructured databases with a system consisting of three main components, supported by two other Stanford projects. See Figure 1. The TDIFF component uses tree differencing algorithms to detect changes between snapshots of semistructured data. The CORE component allows clients to store and query semistructured data and their changes. CORE uses another Stanford database project, *Lore*, to store the data and changes. The QSS component supports powerful and flexible subscriptions over heterogeneous data sources, notifying subscribers of changes of interest. QSS exploits the *Tsimmis* project at Stanford, which gives us the ability to wrap and integrate diverse heterogeneous sources.

Now we use our weather scenario to roughly illustrate how the components work together. The TDIFF component is used to detect relevant changes (such as changes in rainfall measure). CORE can store these changes and clients can query the history, including asking questions of the form: "has the overnight rainfall changed in the past week?" Clients wishing to be notified automatically of certain changes, such as trends in overnight snowfall, may do so by registering a subscription with the QSS component. QSS automatically polls the source data using Tsimmis, detects changes using TDIFF, and stores and queries these changes appropriately using CORE.

5 Overview of System Infrastructure and Components

The supporting Tsimmis project focuses on the integration of heterogeneous data and the use of a standard data model and query language over many different types of sources. A Tsimmis wrapper presents the data from a source in a uniform graph-based data model called the *Object Exchange Model* (OEM). Queries posed over the (virtual) OEM representation of

the source data are translated to native source-specific queries, and the results of these queries are translated back to an OEM representation before being returned to the wrapper client. Wrappers for a variety of data sources can be built quickly using template-based wrapper-generators. In addition, Tsimmis allows clients to access (virtually) integrated data from multiple heterogeneous data sources through its *mediators*. Wrappers and mediators provide identical interfaces to clients.

Lore (for Lightweight Object Repository) is a database system designed to store and query semistructured data in OEM. The Lore system offers the functionality of a traditional DBMS including OQL-like queries, multi-user support and recovery, as well as non-traditional functionality such as path expression matching and extensive type coercion. The *Lorel* query language is used to query Lore data, and can be viewed as an extension to OQL.

5.1 TDIFF: Detecting Changes

The TDIFF module is used to detect changes in the heterogeneous semistructured databases being monitored by our system. In conventional databases, detecting changes to data is made easier by the availability of facilities such as transaction logs, triggers, etc. However, in heterogeneous data sources, such as Web sites, such facilities often are absent. Even in cases where these facilities are available, they may not be accessible. Therefore, in practice, we often need to detect changes by comparing two or more snapshots of the database (or a portion thereof) using differencing algorithms. Here, the semistructured nature of the data causes problems, since finding and representing changes in semistructured data is much harder than in, say, structured relational tables. We have developed algorithms to detect changes between snapshots of tree-structured data, and the TDIFF component is an implementation of these algorithms. . Since the results of Tsimmis queries over wrapped sources are tree-structured, TDIFF is well-suited for detecting changes in any wrapped (or mediated) source.

5.2 CORE: Change Object Repository

While TDIFF gives us the ability to detect changes between two snapshots of a semistructured data source, we would also like the ability to store, browse, and query this temporal data. Existing temporal models do not support the irregularity and incompleteness of semistructured data, so we have developed the *Delta-OEM* (DOEM) data model. DOEM extends OEM by allowing *annotations* on the nodes and edges in the graph representation of OEM. These annotations represent the history of the node or edge, and collectively represent the complete history of the database. Nodes can have *create* and *update* annotations, storing the time of the modification and, for updates, the old and new values of the object. Edges can have *add* and *remove* annotations, storing the time of the modification. The *Chorel* (*Change Lorel*) query language extends the functionality of Lorel, allowing a user to query DOEM data, accessing both historical and current data.

The CORE component is our implementation of the DOEM data model and the Chorel query language. We have implemented them as an extension to Lore. DOEM data is encoded as OEM data (by representing annotations as special OEM objects) and is stored in a Lore database. Chorel queries are then translated to Lorel queries over the OEM-encoded DOEM database. Finally, the encoded results of the Lorel query are translated back into their DOEM representation before being returned to the user. CORE offers a graphical user interface that allows users to issue Chorel queries and browse query results and their histories.

5.3 QSS: Query Subscription Service

A flexible system for detecting and reporting changes must offer a number of options and parameters. For example, one can detect changes to Web sites in a number of ways. Some Web sites offer users the option of receiving e-mail when significant changes are made. Others may need to be polled, and the polling interval may depend on the context. For example, a traffic site is best polled every few minutes, while a site with daily ski reports is best polled once every morning. For sites that change infrequently, the changes may best be detected by user request. Users may also wish to learn about changes in a number of ways. Some users may wish to be notified whenever changes of a certain kind occur. Others may wish to receive a daily or weekly report of changes of interest. Still others may wish to receive a report at their explicit request only. Our QSS component ties together these diverse options in a general-purpose subscription framework.

Each QSS *subscription* consists of three main components. The first component, *the polling query*, is the query that is executed over the information source to gather data of interest. We use Tsimmis queries as polling queries. The second component, the *filter query*, is executed over the history of the data gathered from the source and the results are returned to the subscriber. We use Chorel queries as the filter queries. The third component is a *frequency specification* that determines when and how the polling and filter queries are executed.

5.4 System Architecture

Figure 2 depicts the architecture of our change management system. The QSS component has a client-server architecture, with one or more client processes (*Query Subscription Clients* or QSCs) connected to the server process (QSS). Users interact with the QSC through a graphical user interface, creating subscriptions, issuing probes, and receiving results. The QSS component issues polling queries over information sources (via Tsimmis wrappers or mediators) and the result of each query is stored by QSS as the current snapshot for this query. This snapshot and the previous snapshot are sent to the TDIFF component, which identifies the changes to the data and stores them within CORE. QSS issues filter queries to the CORE database and the result of each query, if non-empty, is returned to the QSS component. QSS then sends the changes to the appropriate QSC clients, which notify their

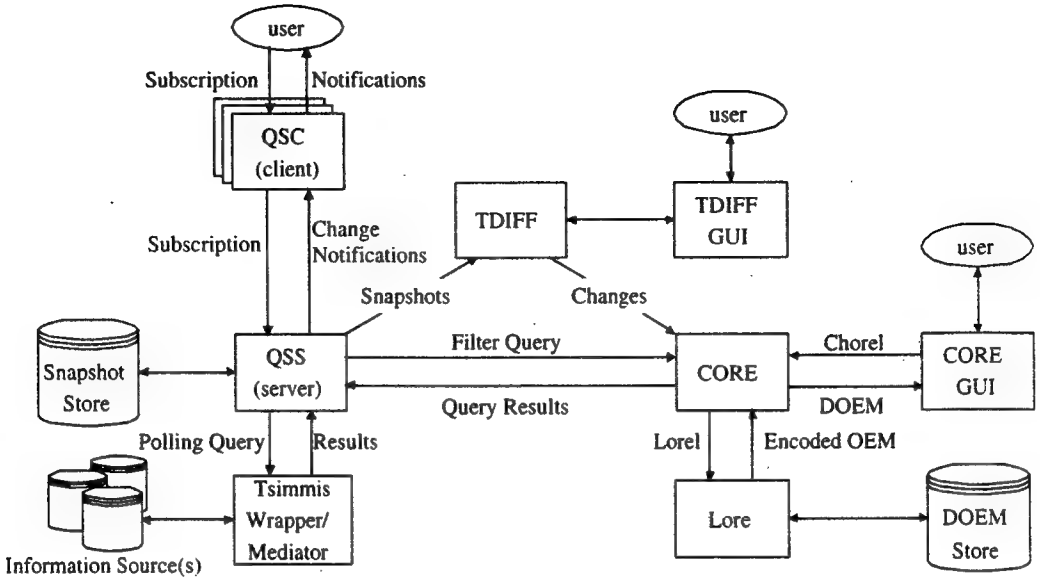


Figure 2: System Architecture

subscribers about the fresh results of their subscriptions. Users can view subscription results using the QSC user interface. In addition to the subscription service, our change management system offers users the ability to interact directly with TDIFF or with CORE, which is useful for independently exploiting their functionality.

6 Demonstration Walk-Through

Our change management system is fully implemented, and a system demonstration "walk-through" is now provided. We will continue with our running example of a Web-based weather site that offers information such as temperature, precipitation, weather advisories, and road conditions for many cities. This site is updated frequently as weather conditions change.

The first part of our demonstration shows the detection of changes in the data source by the TDIFF component. For example, we can load two snapshots of the weather source into TDIFF and it will tell us, e.g., whether the snowfall for an area has increased or decreased and if so, what the old and new values are. Using the TDIFF graphical user interface, we can browse these changes and observe that the overnight snowfall of a city has gone from 3 inches to 0 inches, probably indicating the end of a storm. Further examples of TDIFF can be found at `http://www-db.stanford.edu/c3/`.

In the next part of our demo, we illustrate how CORE is used to browse and query the history of the weather site. In particular, we run TDIFF on a series of snapshots of the Web site and load the detected changes into a CORE database. The CORE user interface allows us

to browse the complete history of the data, including updates that were made months ago. For example, we may observe that Tahoe City went through a major heat-wave last summer. with temperatures ranging from 90 to 100 degrees during August. In addition to browsing the data's history, we can also pose expressive Chorel queries over the data through the CORE user interface. For example, we can submit a query asking for "all California cities where the overnight snowfall has decreased by at least 6 inches" to determine if a storm is subsiding or not. This query would be expressed in Chorel as follows. (Like other database query languages, we expect Chorel queries to be issued primarily by client software interfacing with user-friendly GUI tools.)

```
select C
from Weather.CA.city C
where exists S in C.snowfall<upd at T from OldVal to NewVal> :
      (OldVal - NewVal) > 6
```

We can then use the CORE interface to browse the results. The interface displays the cities and their subobjects (temperature, snowfall, etc.) in a graphical format, and we can expand the history of any edge or node in the display (e.g., the snowfall value for each city). This expansion shows all updates to this value, and we can determine if the decline in snowfall is a continuing trend.

The final part of the demonstration shows how the results of the above query can be monitored automatically using QSS. Avid skiers can create a subscription and be notified automatically anytime there has been a good snow-storm that seems to be clearing up. suggesting good skiing conditions. Using the QSS user interface, we create a subscription over the weather source. The polling query retrieves all information about California cities:

```
select C
from Weather.CA.city C
```

The filter query is the Chorel query shown earlier. We set the frequencies to poll and filter once a day. Whenever the snowfall of a city in California decreases by six inches, QSS will notify us. When we are notified of the update, we can log into the QSS interface and see the result of the filter query, which tells us which cities have had large snow-storms subside.

7 Further Details

In the remainder of the report we provide additional examples and more in-depth coverage of the C^3 data model, query language, and system implementation.

7.1 Additional Motivating Examples

The *Palo Alto Weekly*, a local newspaper, maintains a Web site providing information about restaurants in the Bay Area. Most of the data in the restaurant guide is relatively static. But

as often happens in database applications, we are particularly interested in the dynamic part of the data. For example, we are interested in finding out which restaurants were recently added, which restaurants were seen as improving, degrading, etc. These changes can be captured by a tool that we have implemented, called TDIFF. The TDIFF program takes two versions of a web page as input, and produces as output a marked-up copy of the web page that highlights the differences between the two versions based on their semistructured contents. Our TDIFF system allows users to browse the marked-up web page to view the changes, and to travel back and forth between the old and new versions of the document.

For reasonably small documents, browsing the marked-up HTML files produced by TDIFF to view the changes of interest is a feasible option. However, as documents get larger and changes become more prevalent and varied, one soon feels the need to use queries to directly find changes of interest instead of simply browsing. (For example, the restaurant guide page is currently more than 20,000 lines long, making browsing very inconvenient.) An example of a simple change query over the restaurant data is "find all new restaurant entries." Another example is "find all restaurants whose average entree price changed." Just as browsing databases is often an ineffective way to retrieve information, the same holds for browsing data representing changes. Thus, for this example, what we need is a query language that allows queries over changes to (semistructured) HTML pages.

As another motivating example, consider a typical information library system that contains book circulation information. Suppose we wish to be notified whenever any "popular" book becomes available where, say, we define a book as popular if it has been checked out two or more times in the past month. We could partially achieve this goal by setting a trigger on the circulation database that notifies us whenever a book is returned. However, there are two problems with this approach. First, many library information systems are legacy mainframe applications on which triggers are not available. Furthermore, even in cases where the library information system is implemented using a database system that supports triggers, a user often lacks the access rights required to set triggers on the database. Second, there is often no way to access historical circulation information, so that we cannot check whether the book being returned was checked out two or more times recently. In this application too, the data may be semistructured, especially if the library system merges information from multiple sources. Thus, we again need a method to compute, represent, and query changes in the context of semistructured data.

7.2 The Object Exchange Model

The *Object Exchange Model* (OEM) is a simple, flexible model for representing heterogeneous, semistructured data. (Recall that semistructured data is data that may be irregular or incomplete, and that does not necessarily conform to a fixed schema, e.g., HTML documents describing restaurants.) In this subsection, we begin by briefly describing OEM. Next, we define the basic change operations used to modify an OEM database. Finally, we introduce the concept of an OEM *history* that describes a collection of basic change operations. Histories

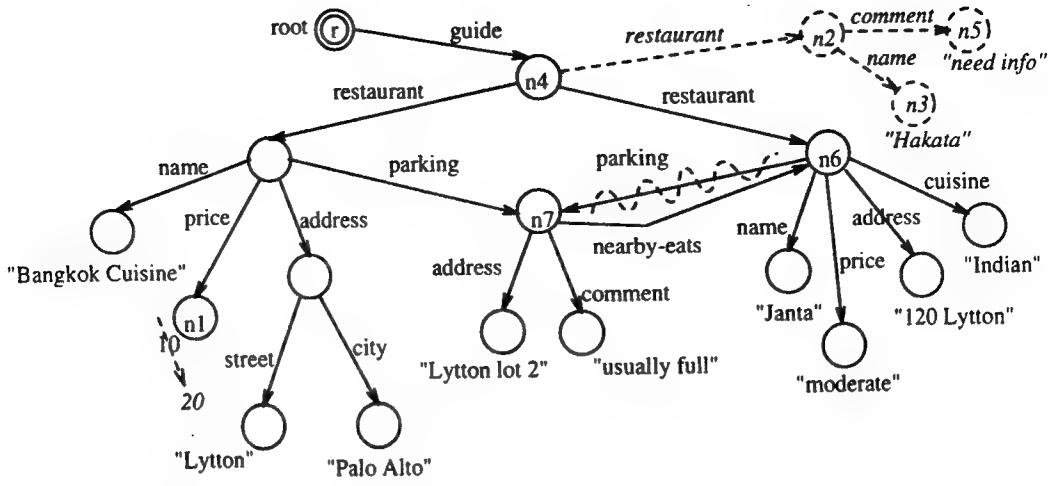


Figure 3: The OEM database in Examples 7.1 and 7.2

form the basis of our change representation model described in Subsection 7.3.

Intuitively, one can think of an OEM database as a graph in which nodes correspond to objects and arcs correspond to relationships. Each arc has a label that describes the nature of the relationship. (Note that the graph can have cycles, and that an object may be a subobject of multiple objects via different relationships. Example 7.1 below illustrates these points.) Nodes without outgoing arcs are called *atomic objects*; the rest of the nodes are called *complex objects*. Atomic objects have a *value* of type integer, real, string, etc. An arc (p, l, c) in the graph signifies that the object with identifier c is an l-labeled subobject (child) of the complex object with identifier p. Each OEM database has a distinguished node called the *root* of the database. The root is the implicit starting point of path expressions in the Lorel query language (described in Subsection 7.4.1). Formally, we define an OEM database as follows:

Definition 7.1 An OEM *database* is a 4-tuple $O = (N, A, v, r)$, where N is a set of object identifiers; A is a set of labeled, directed arcs (p, l, c) where $p, c \in N$ and l is a string; v is a function that maps each node $n \in N$ to a value that is an integer, string, etc., or the reserved value C (for complex); and r is a distinguished node in N called the *root* of the database. A node is a *complex object* if its value is C and otherwise it is an *atomic object*. Only complex objects have outgoing arcs. We also require that every node be reachable from the root using a directed path. □

Example 7.1 We will use as our running example an OEM database describing the restaurant guide subsection of the *Palo Alto Weekly*, introduced earlier. Figure 3 shows a small portion of the data. (For this example, ignore items depicted using dashed lines.) Note that although the restaurant entries are quite similar to each other in structure, there are important differences that require the use of a semistructured data model such as OEM. In

particular, we see that the price rating for a restaurant may be either an integer (10) or a string ("moderate"). The address may be either a simple string ("120 Lytton") or a complex object with subobjects listing the street, city, etc. Note also that although the data has a natural hierarchical structure, nodes may have multiple incoming arcs (e.g., node n_7), and there are cycles (e.g., the cycle formed by the arcs "parking" and "nearby-eats"). In the sequel, we refer to this database as *Guide*. □

7.2.1 Changes in OEM

We now describe how an OEM database is modified. Let $O = (N, A, v, r)$ be an OEM database. The four *basic change operations* are the following:

Create Node: The operation $creNode(n, v)$ creates a new object. The identifier n must be new, i.e., n must not occur in O. The initial value v must be an atomic value (integer, real, string, etc.) or the special symbol C (for complex).

Update Node: The operation $updNode(n, v)$ changes the value of object n, where v is an atomic value or the special symbol C. Object n must be either an atomic object or a complex object without subobjects. (The model requires us to remove all subobjects of a complex object n before transforming it into an atomic object.) The value v becomes the new value of n.

Add Arc: The operation $addArc(p, l, c)$ adds an arc labeled l from object p (the parent) to object c (the child). Objects p and c must exist in O, p must be complex, and the arc (p, l, c) must not already exist in O.

Remove Arc: The operation $remArc(p, l, c)$ removes an arc. Objects p and c must exist in O, and O must contain the arc (p, l, c), which is removed.

If u is a basic change operation that can be applied to O, we say u is *valid* for O, and we use $u(O)$ to denote the result of applying u to O. Note that there is no explicit object deletion operation. In OEM, persistence is by reachability from the distinguished root node. Thus, to delete an object it suffices to remove all arcs leading to it. A subtlety is that sometimes we need to allow objects to be "temporarily" unreachable. In particular, when we create a new object, it remains unreachable until we create an arc that links it to the rest of the database. Thus, when we consider sequences of changes in Subsection 7.2.2, we want to permit the result of atomic changes to (temporarily) contain unreachable objects. The issue is discussed further in Subsection 7.2.2 below. Note that users will typically request "higher-level" changes based on the Lorel update language; the basic change operations defined here reflect the actual changes at the database level.

Example 7.2 Let us consider some modifications to the OEM database in Example 7.1. We will use these modifications as a running example in the rest of this section. First, on January 1st, 1997, the price rating for "Bangkok Cuisine" is changed from 10 to 20. This modification corresponds to an *updNode* operation. On the same day, a new restaurant with name "Hakata" is added (with no other data). This modification corresponds to two

creNode operations for the restaurant node and its subobject, and two *addArc* operations to add arcs labeled "restaurant" and "name." Next, on January 5th, a subobject with value "need info" is added to the "Hakata" restaurant object via an arc labeled "comment." This corresponds to one *creNode* operation and one *addArc* operation. Finally, on January 8th the parking at "Lytton lot 2" is no longer considered suitable for the restaurant "Janta," and the corresponding arc is removed; this modification corresponds to a *remArc* operation. These changes are depicted in Figure 3 using dashed lines.

7.2.2 OEM Histories

We are typically interested in collections of basic change operations, which describe successive modifications to the database. We say that a *sequence* $L = u_1, u_2, \ldots, u_n$ of basic change operations is *valid* for an OEM database O if u_i is valid for O_{i-1} for all $i = 1 \ldots n$, where $O_0 = O$, and $O_i = u_i(O_{i-1})$, for $i = 1 \ldots n$. We use $L(O)$ to denote the OEM database obtained by applying the entire sequence L to O. Also, we say that a *set* $U = \{u_1, u_2, \ldots, u_n\}$ of basic change operations is *valid* for an OEM database O if (1) for some ordering L of the changes in U, L is a valid sequence of changes, (2) for any two such valid sequences L and L', $L(O) = L'(O)$, and (3) U does not contain both $addArc(p, l, c)$ and $remArc(p, l, c)$ for any p, l, and c. We use $U(O)$ to denote the OEM database obtained by applying the operations in the set U (in any valid order) to O.

We are now ready to define an OEM history. Assume we are given some time domain **time** that is discrete and totally ordered; elements of **time** are called *timestamps*. Intuitively, consider an OEM database to which, at some time t_1, a set U_1 of basic change operations is applied, then at a later time t_2, another set U_2 is applied, and so on. A history represents such a sequence of sets of modifications.

Definition 7.2 An OEM *history* is a sequence $H = (t_1, U_1), \ldots, (t_n, U_n)$, where U_i is a set of basic change operations and t_i is a timestamp, for $i = 1 \ldots n$, and $t_i < t_{i+1}$ for $i = 1 \ldots n-1$. We say H is *valid* for an OEM database O if, for all $i = 1 \ldots n$, U_i is valid for O_{i-1}, where $O_0 = O$, and $O_i = U_i(O_{i-1})$ for $i = 1 \ldots n$. □

We now return to the requirement that all objects in an OEM database must be reachable from the root. An OEM history can be viewed as a sequence $L_1, ..., L_n$ of sequences of atomic changes. Within one sequence L_i of changes, we relax the requirement that all objects are reachable from the root so that we can, e.g., create a node and then create arcs leading to it, as discussed earlier. However, immediately after each sequence L_i has been applied, nodes that are unreachable are considered as deleted, and the remainder of the history should not operate on these objects. To simplify presentation, we also assume that object identifiers of deleted nodes are not reused.

Example 7.3 The history for the modifications described in Example 7.2 consists of three sets of basic change operations. It is given by $H = ((t_1, U_1), (t_2, U_2), (t_3, U_3))$, where $t_1 =$

1Jan97, t_2 = *5Jan97*, t_3 = *8Jan97*, and:

$$
\begin{aligned}
U_1 = \{\ & updNode(n_1, 20),\ creNode(n_2, \mathcal{C}), \\
& creNode(n_3, \text{“}Hakata\text{”}),\ addArc(n_4, \text{“}restaurant\text{”}, n_2), \\
& addArc(n_2, \text{“}name\text{”}, n_3)\ \} \\
U_2 = \{\ & creNode(n_5, \text{“}need\ info\text{”}) \\
& addArc(n_2, \text{“}comment\text{”}, n_5)\ \} \\
U_3 = \{\ & remArc(n_6, \text{“}parking\text{”}, n_7)\ \}.
\end{aligned}
$$

This is a valid history for the original OEM database in Figure 3. □

7.3 Representation of Changes

In this subsection, we describe how changes to an OEM database are represented by attaching *annotations* to the OEM graph, thereby turning it into a DOEM (*Delta* OEM) graph. Intuitively, annotations are tags attached to the nodes and arcs of an OEM graph that encode the history of basic change operations on those nodes and arcs. There is a one-to-one correspondence between annotations and the basic change operations. Thus, nodes and arcs may have the following four types of annotations: (1) $cre(t)$: the node was created at time t. (2) $upd(t, ov)$: the node was updated at time t; ov is the old value. (3) $add(t)$: the arc was added at time t. (4) $rem(t)$: the arc was removed at time t. The set of all possible node annotations is denoted by **node-annot**, and the set of all possible arc annotations is denoted by **arc-annot**.

Using the above definitions of node and arc annotations, we now define a DOEM database. In the following definition, the function $f_N(n)$ maps a node n to a set of annotations on that node and the function $f_A(a)$ maps an arc a to a set of annotations on that arc.

Definition 7.3 A DOEM *database* is a triple $D = (O, f_N, f_A)$, where $O = (N, A, v, r)$ is an OEM database, f_N maps each node in N to a finite subset of **node-annot**, and f_A maps each arc in A to a finite subset of **arc-annot**. □

7.3.1 DOEM Representation of an OEM History

Given an OEM database O and a history $H = (t_1, U_1), ..., (t_n, U_n)$ that is valid for O, we would like to construct the DOEM *database representing* O *and* H, denoted by $D(O, H)$. $D(O, H)$ is constructed inductively as follows. We start with a DOEM database D_0 that consists of the OEM database O with empty sets of annotations for the nodes and the arcs of O. Suppose D_{i-1} is the DOEM database representing O and $(t_1, U_1), ..., (t_{i-1}, U_{i-1})$, for some $1 \leq i \leq n$. The DOEM database D_i is constructed by considering the basic change operations in U_i. Since the history is valid, we can assume some ordering L_i of the operations in U_i (Definition 7.2). Starting with D_{i-1}, we process the operations in L_i in order. Whenever the value of an object is updated, in addition to performing the update we attach an *upd*

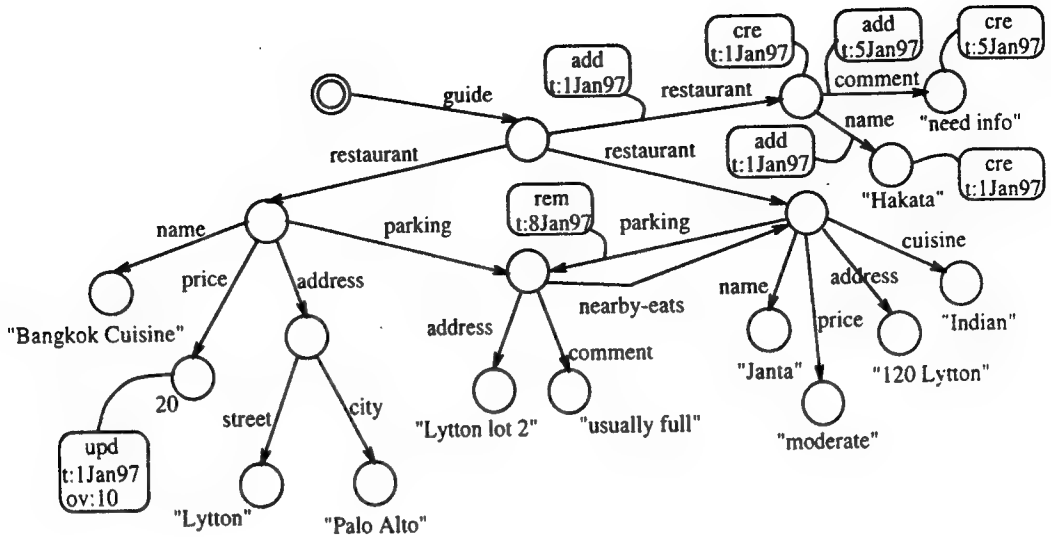


Figure 4: The DOEM database in Example 7.4.

annotation to the node. This annotation contains the timestamp t_i and the old value of the object. When a new object is created or an arc added, in addition to performing the modification, we attach a *cre* or *add* annotation with the timestamp t_i. When an existing arc is removed, we do not actually remove the arc from the graph; instead, we simply attach a *rem* annotation to the affected arc with the timestamp t_i.

Example 7.4 Consider the history described in Example 7.3, which transforms the OEM database of Figure 3 as depicted there using dashed lines. The corresponding DOEM database is shown in Figure 4. We see that the DOEM database contains several annotations, depicted as boxes in the figure. For example, the annotations with timestamp "1Jan97" correspond to the first set of updates. Note that the *cre*, *add*, and *rem* annotations contain only the timestamp, while the *upd* annotation also contains the old value of the updated node (10, in our example). Also note that the removed "parking" arc from the "Janta" restaurant object to the "Lytton lot 2" parking object is not actually removed from the DOEM database; instead it bears a *rem* annotation. □

7.3.2 Properties of DOEM Databases

We now summarize the desirable properties of the DOEM representation of OEM database histories. Given a DOEM database D, it is easy to obtain the *original snapshot*, $O_0(D)$, the *snapshot at time t*, $O_t(D)$, and, the *current snapshot*, $O_c(D)$. It is also easy to obtain the *encoded history* $H(D)$ from a DOEM database D. We say that a DOEM database D is *feasible* if there exists some OEM database O and valid history H such that $D = D(O, H)$. Note that we do not require DOEM databases to record all changes since creation, i.e., OEM database O need not be empty. It is relatively easy to determine if a given DOEM database D is

feasible. Given a feasible DOEM database D, we can show that the OEM database $O_0(D)$ and the history $H(D)$ encoded by D are unique. Thus, a DOEM database faithfully captures all the information about the history of the corresponding OEM database. Finally, as we will see in the next subsection, it is easy and intuitive to query the history encoded in a DOEM database.

7.4 Querying Over Changes

In Subsection 7.3, we have seen how the history of an OEM database is represented by the corresponding DOEM database. In this subsection, we describe how DOEM databases are queried. We introduce a query language called *Chorel* for this purpose. Chorel is similar to the Lorel language used to query OEM databases. We begin with a brief overview of Lorel, followed by a description of the syntax and semantics of Chorel.

7.4.1 Lorel Overview

Lorel uses the familiar *select-from-where* syntax, and can be thought of as an extension of OQL in two major ways. First, Lorel encourages the use of path expressions. For instance, one can use the path expression *guide.restaurant.address.street* to specify the streets of all addresses of restaurant entries in the Guide database. Second, in contrast to OQL, Lorel has a very "forgiving" type system. When faced with the task of comparing different types, Lorel first tries to coerce them to a common type. When such coercions fail, the comparison simply returns false instead of raising an error. This behavior, while it may be unsuitable for traditional databases, is exactly what a user expects when querying semistructured data. Lorel also provides a number of syntactic conveniences such as the possibility of omitting the *from* clause. We do not describe Lorel in detail here, but only present through a simple example those features that are needed to understand Chorel.

Example 7.5 Consider again the (modified) OEM database depicted in Figure 3. To find all restaurants that have a price rating of less than 20.5, we can use the following Lorel query:

```
select guide.restaurant
where guide.restaurant.price < 20.5
```

Note that the query expresses the price rating as a real number whereas the restaurant entries for "Bangkok Cuisine" and "Janta" in the modified OEM database shown in Figure 3 use an integer and a string, respectively. Furthermore, the third restaurant entry does not have a price subobject at all. Lorel successfully coerces the integer price 10 to real, and the comparison succeeds. For the string encoding of the price ("moderate"), Lorel tries to coerce, but fails, returning false as the result of the comparison. Finally, for the third restaurant, the missing price subobject simply causes the comparison to return false. Thus, the result of the above query is a singleton set containing the restaurant object for "Bangkok Cuisine."

Note that this is an intuitively reasonable response to the original query, despite the typing difficulties and the missing data. □

7.4.2 Chorel

In Chorel, path expressions may contain *annotation expressions*, which allow us to refer to the node and arc annotations in a DOEM database. Informally, Lorel path expressions can be thought of as being matched to paths in the OEM database during query execution. Analogously, the annotation expressions in Chorel path expressions can be thought of as being matched to annotations on the corresponding paths in the DOEM database.

Example 7.6 Consider the DOEM database depicted in Figure 4. To find all newly added restaurant entries only, we can use the following Chorel query:

```
select guide.<add>restaurant
```

The annotation expression "¡*add*¿" specifies that only those objects connected to the "guide" object by a "restaurant"-labeled arc having an *add* annotation should be retrieved. For the database depicted in Figure 4, this Chorel query returns the restaurant object with name "Hakata." □

Not surprisingly, we use four kinds of annotation expressions in Chorel path expressions: *node annotation expressions* "*cre*" and "*upd*," and *arc annotation expressions* "*add*" and "*rem*." Recall that a path expression, e.g., *guide.restaurant.price*, consists of a sequence of labels. Arc annotation expressions must occur immediately before a label, whereas node annotation expressions must occur immediately after one. (Note that since node and arc annotations use different keywords, no confusion can arise.) Path expressions containing node or arc annotation expressions are called *annotated path expressions*. For instance, *guide.¡add¿restaurant.price¡upd¿* is a correct annotated path expression. It requires an *add* annotation to be present on the arc labeled "restaurant," and an *upd* annotation on the "price" node (i.e., on the node at the destination of the arc labeled "price"). For simplicity, we do not consider path expressions that have annotation expressions attached to wildcards or regular expressions, however generalizing to allow such annotation expressions is not difficult.

Annotation expressions may also introduce *time variables* to refer to the timestamps stored in matching annotations, and *data variables* to refer to the modified values in matching *upd* annotations. More precisely, the syntax of annotation expressions is as follows:

¡*Annot* [*at timeV*]¿ if *Annot* is in { *add*, *rem*, *cre* }
¡*upd* [*at timeV*] [*from oldV*] [*to newV*]¿ for *upd*

where *timeV*, *oldV*, and *newV* are variables. Note that a DOEM database does not explicitly store the new value of an updated object, however this information is available implicitly, and can be determined easily.

Example 7.7 Consider the DOEM database in Figure 4. To find all restaurant entries that were added before January 4th, 1997, we can use the following Chorel query:

```
select guide.<add at T>restaurant
where T < 4Jan97
```

The Chorel preprocessor will rewrite this query to obtain the following. (We will explain this rewriting shortly.)

```
select R
from guide.<add at T>restaurant R
where T < 4Jan97
```

The introduced *from* clause will bind R to all "restaurant" objects that are connected to the "guide" object via an arc with an *add* annotation, and will provide corresponding bindings for T. More precisely, the evaluation of the *from* clause will yield the set of pairs $\langle R, T \rangle$ such that there is a *restaurant* arc from the *guide* object to R that has an *add* annotation with timestamp T. The *where* clause will filter out the $\langle R, T \rangle$ pairs for which T does not satisfy the condition. For the DOEM database in Figure 4, this query returns the restaurant object for "Hakata." □

Once time and data variables have been bound using annotations, they can be used just like other variables in Lorel or OQL. This is illustrated by the following query, which uses time and data variables in the *select* clause.

Example 7.8 Referring again to the DOEM database in Figure 4, suppose we want to find the names of all restaurants whose price ratings were updated on or after January 1st, 1997 to a value greater than 15, together with the time of the update and the new price. We can use the following query:

```
select N, T, NV
from guide.restaurant.price<upd at T to NV>,
          guide.restaurant.name N
where T >= 1Jan97 and NV > 15

answer
   name "Bangkok Cuisine"
   new-value 20
   update-time 1Jan97
```

The result of the above query is a single complex object with three components, as shown above. The label *name* is chosen by Chorel. For time and data variables whose labels are not specified by the query, Chorel chooses the default labels *create-time, add-time, remove-time, update-time, new-value*, and *old-value*. □

7.4.3 Chorel Semantics

We now make the semantics of Chorel queries more precise. As is done for Lorel, the semantics is described by specifying the rewriting of Chorel queries into OQL-like queries. However, we need to introduce some additional machinery to handle the annotation expressions in Chorel queries.

First, the annotation expressions in a Chorel query are transformed into a canonical form that includes all variables. For example, "*¡add¿*" is rewritten to "*¡add at T1¿*," and "*¡upd from X¿*" is rewritten to "*¡upd at T2 from X to NV2¿*," where *T1*, *T2*, and *NV2* are fresh variables. Next, as in Lorel, we eliminate path expressions by introducing variables for the objects "inside" the path expressions. For example, the path expression "*a.b.c*" in a *from* clause is converted to "*a.b X, X.c Y*," where *X* and *Y* are new *range variables.*

At this stage, we have to give a semantics to range variable definitions that may include annotation expressions (e.g., "*X.lab Y*," "*X.¡add at T¿lab Y*") in the context of a DOEM database. In the absence of an annotation expression, the semantics of an expression "*X.lab Y*" is that for a binding o_X of *X*, *Y* is bound to all objects o_Y such that there is an arc labeled *lab* from o_X to o_Y in the current snapshot. Note that by this semantics, a standard Lorel query (without annotations) over a DOEM database has exactly the semantics of the same query asked over the current snapshot for that DOEM database. In the presence of annotation expressions, the semantics requires the existence of the specified annotation, and also provides bindings for the variables in the annotation expression. The bindings are also specified by a special rewriting. As an example, the query in Example 7.8 is rewritten to:

```
select N, T, NV
from guide.restaurant R, R.price P, R.name N,
     (T, OV, NV) in updFun(P)
where T >= 1Jan97 and NV > 15
```

Our rewriting uses the following functions, which extract the information stored in annotations:

$$creFun(node) \rightarrow \{time\}$$
$$updFun(node) \rightarrow \{(time, old\text{-}value, new\text{-}value)\}$$
$$addFun(source, label) \rightarrow \{(time, target)\}$$
$$remFun(source, label) \rightarrow \{(time, target)\}$$

The function *creFun(n)* returns the set of timestamps found in *cre* annotations on node *n*. (Note that by our definition of change operations in Subsection 7.2.1, this set is either empty or a singleton.) The function *updFun(n)* returns a set of triples corresponding to the timestamp, the old value, and the new value in *upd* annotations on *n*. The function *addFun(n,l)* returns a set of (t, c) pairs such that *c* is an *l*-labeled subobject of *n* via an arc that has an $add(t)$ annotation. The *remFun* function is analogous to *addFun*. Once this rewriting has been performed, the *from*, *where*, and *select* clauses of the resulting query are processed in a standard manner.

Above, we have illustrated how variables introduced in the *from* clause are interpreted. Variables may be introduced in the *where* clause as well. They are treated by introducing existential quantification in the *where* clause, extending the treatment of such variables in Lorel. Consider the following example:

Example 7.9 Consider again the DOEM database of Figure 4. Suppose we want the names of restaurants to which a "moderate" price subobject was added since January 1st, 1997. We can write the following Chorel query:

```
select N
from guide.restaurant R, R.name N
where R.<add at T>price = "moderate" and T >= 1Jan97
```

The variable T is introduced in the *where* clause. Therefore, the rewritten *where* clause is:

```
where exists (T, P) in addFun(R,"price") :
      (P = "moderate" and T >= 1Jan97)
```

□

7.5 Implementing DOEM and Chorel

In this subsection, we describe how we implement DOEM databases and Chorel queries. We encode DOEM databases as OEM databases, and we implement Chorel by translating Chorel queries to equivalent Lorel queries over the OEM encoding of the DOEM database. In addition to being more modular than a direct implementation approach that builds a Chorel database engine from scratch, this approach can also be adapted easily to other graph-based data models.

7.5.1 Encoding DOEM in OEM

Let D be a DOEM database. We encode D as an OEM database O_D defined as follows. For each object o in D, there is a corresponding object o' in O_D. An atomic object is encoded as a complex object so that we can record its history. Special labels used by the encoding start with the special character "*&*" to distinguish them from standard labels occuring in O. The encoding object o' has the following subobjects, listed by their labels.

- *&val*: If o is atomic with current value v, there is a "*&val*"-labeled arc from o' to an atomic object with value v. If o is complex, there is a "*&val*"-labeled arc from o' to itself. (This extra edge simplifies the translation of Chorel queries to equivalent Lorel queries over the encoding.)
- *&cre*: If o has a create annotation $cre(t)$, then o' has a "*&cre*"-labeled atomic subobject with value t.

- *&upd*: For each update annotation $upd(t, ov)$ attached to o, o' has an "*&upd*"-labeled complex subobject with the following structure: a "*&time*"-labeled subobject with value t, an "*&ov*"-labeled subobject with the value before the update (ov), and a "*&nv*"-labeled subobject with the value after the update.
- l: If the current snapshot for D contains an arc (o, l, p), then O_D contains an arc labeled l from o' to the object p' that encodes p.
- *&l-history*: If D contains an arc (o, l, p), then O_D contains an arc $(o', \&l\text{-}history, o'_l)$ where o'_l is a complex object that contains the history of the l arcs from o to p. The object o'_l has the following structure: (1) *&target*: There is an arc $(o'_l, \&target, p')$. where p' is the object encoding p. (2) *&add*, *&rem*: For each annotation $add(t)$ ($rem(t)$) attached to (o, l, p), there is an "*&add*"-labeled (respectively, "*&rem*"-labeled) atomic subobject with value t.

It can be shown that all the information in a DOEM database D is fully represented in D's OEM encoding using the above scheme.

7.5.2 Translating Chorel to Lorel

Given the above encoding of a DOEM database as an OEM database, we now describe how a Chorel query over a (conceptual) DOEM database is translated into an equivalent Lorel query over an OEM encoding of the DOEM database. The following example intuitively presents the basis of the translation scheme.

Example 7.10 Consider the Chorel query in Example 7.9. In Subsection 7.4.3, we considered the OQL-like rewriting of this query. We now complete this rewriting by using the information encoded in the &-arcs to yield the following Lorel query over the OEM encoding of the DOEM database in Figure 4:

```
select N
from guide.restaurant R, R.name N
where exists H in R.&price-history :
      exists P in H.&target :
      exists T in H.&add :  T >= 1Jan97 and
            P.&val = "moderate"
```

Note that we simulate the range specification *addFun(R,"price")* using the "&"-prefixed subobjects. Further, we use *P.&val* to access the actual price value (and not the complex object packaging it with its history). □

Note that the previous query returns a set of DOEM objects that represent restaurant names. That is, it returns not only the names of the restaurants, but also the history of these names, if they changed. Returning the DOEM object enables a user interface to access both the value and the history of an object. We have implemented a DOEM database system, called CORE, based on the above ideas.

7.6 A Query Subscription Service

Earlier we mentioned an important application of change management: being able to notify "subscribers" of changes in (semistructured) information sources of interest to them. In this subsection, we describe the design and implementation of such an application, called a *Query Subscription Service* (QSS), using DOEM and Chorel.

An ordinary query is evaluated over the current state of the database, the results passed to the client and then discarded. An example of an ordinary query is "find all restaurants with Lytton in their address." In contrast, a *subscription query* is a query that repeatedly scans the database for new results based on some given criteria and returns the changes of interest. An example of a subscription query is "every week, notify me of all *new* restaurants with Lytton in their address." Below, we describe how subscription queries are specified and implemented in our system.

Supporting subscription queries introduces the following challenges. First, as discussed earlier, many information sources that we are interested in (e.g., library information systems, Web sites, etc.) are *autonomous* and typical database approaches based on triggering mechanisms are not usable. Second, these information sources typically do not keep track of historical information in a format that is accessible to the outside user. Thus, a subscription service based on changes must monitor and keep track of the changes on its own, and often must do so based only on sequences of snapshots of the database states.

Briefly, our approach to constructing a query subscription service over semistructured, possibly legacy information sources is as follows: We access the information sources using Tsimmis *wrappers* or *mediators*, which present a uniform OEM view of one or more data sources. We obtain snapshots of relevant portions of the data, and use differencing techniques to infer changes based on these snapshots. Finally, we use DOEM to represent the changes, and Chorel to specify the changes of interest. We describe our approach in more detail next.

A *subscription* consists of three main components. The first component is a *frequency specification* f that specifies how often QSS should check the information source for data and changes of interest. Examples of frequency specifications are "every Friday at 5:00pm" and "every 10 minutes." The frequency specification implies a sequence of time instants $(t_1, t_2, t_3, \ldots)$, which we call *polling times*. These times are the times when we obtain a new snapshot of the data. (In the actual system we also consider two other modes: one in which the snapshots are obtained following explicit user requests, and the other in which snapshots are obtained as a result of a trigger on the source database firing, if the source provides such a triggering mechanism. To simplify the presentation, we will not consider these modes further here.

The second component of a subscription is a Lorel query Q_l, which we call the *polling query*. QSS sends the polling (Lorel) query to the wrapper or mediator at the polling times $(t_1, t_2, t_3, \ldots)$ to obtain results $(R_1, R_2, R_3, \ldots)$. An example polling query is the following. (In Lorel, "#" is a special character that matches any sequence of zero or more labels in a path, and the operator *like* performs string matching.)

```
define polling query LyttonRestaurants as
    select guide.restaurant
    where guide.restaurant.address.# like "%Lytton%"
```

Let R_0 be the empty OEM database, and let R_i be the result of the polling query on the source at time t_i for $i = 1, 2, \ldots$. Each R_i (a Tsimmis query result) is a tree-structured OEM database. Using differencing techniques, QSS obtains a history $H = (t_1, U_1), (t_2, U_2), \ldots$ corresponding to the sequence of OEM databases $(R_0, R_1, R_2, \ldots)$. That is, $U_i(R_{i-1}) = R_i$ for all $i > 0$. Then, QSS constructs a DOEM database $D(R_0, H)$ corresponding to this history H and the initial snapshot R_0, as described in Subsection 7.3. Thus, intuitively, in the first timestep the results of the polling query are all "created." Thereafter, each subsequent timestep annotates the DOEM database with the changes to the result of the polling query since the previous timestep. We identify the DOEM database corresponding to a polling query using the name of the polling query. Thus the name of the DOEM database corresponding to the above polling query is "*LyttonRestaurants*."

The third component of a subscription is a Chorel query Q_c, called the *filter query*, over the above DOEM database. In Q_c, we can use a special time variable "*t[0]*" to refer to the current polling time t_k. Similarly, we can use "*t[-1]*," "*t[-2]*," etc., to refer to the past polling times t_{k-1}, t_{k-2}, etc., respectively. (If the current polling time is t_k, we define *t[-i]* to be t_{k-i} if $i < k$, and negative infinity otherwise.) The filter query describes the data and changes of interest to the user. An example of an filter query is the following:

```
define filter query NewOnLytton as
    select LyttonRestaurants.restaurant<cre at T>
    where T > t[-1]
```

Given our definition of the DOEM database "*LyttonRestaurants*," this query indicates that the user should be notifed of new restaurants that have Lytton in their address since the last polling time. At each time instant t_k ($k > 0$) specified by the frequency specification, QSS evaluates Q_c over the DOEM database $D(R_0, H_k)$, where $H_k = (t_1, U_1), \ldots, (t_k, U_k)$, and returns the results to the user.

Example 7.11 Consider again the changes to the Guide data described in Example 7.2. Suppose we are interested in being notified every night of new restaurants created in the Guide database since the previous night. We issue the subscription $S = \langle f, Q_l, Q_c \rangle$, where the frequency specification f is "every night at 11:30pm," and the polling query Q_l and filter query Q_c are *Restaurants* and *NewRestaurants* (respectively) as defined below:

```
define polling query Restaurants as
    select guide.restaurant
define filter query NewRestaurants as
    select Restaurants.restaurant<cre at T>
    where T > t[-1]
```

Suppose we create this subscription S on December 30th, 1996, at 10:00am. The polling times given by our frequency specification are t_1 = *30Dec96*, t_2 = *31Dec96*, t_3 = *1Jan97*, and so on (all at 11:30pm). At polling time t_1, QSS sends the polling query Q_l to the Guide OEM database, to obtain the result R_1 consisting of the two original restaurant objects in Figure 3. Since R_0 is the empty OEM database by definition, both restaurant objects will have a *cre* annotation in the DOEM database built by QSS. These annotations all have a timestamp t_1, while the variable *t[-1]* in the query Q_c has value negative infinity at t_1. Therefore, evaluating the filter query Q_c on this DOEM database returns the two restaurant objects as the initial results to the user.

At polling time t_2, the Guide database is unchanged, so the result R_2 of the polling query is identical to R_1. Consequently, no changes are made to the DOEM database maintained by QSS. Note also that at time t_2, *t[-1]* $= t_1$, so that the create annotations on the restaurant objects in the DOEM database no longer satisfy the predicate $T >$ *t[-1]* in the where clause of Q_c. Therefore, the result of Q_c is empty, and the user does not receive any notification.

Before polling time t_3, the Guide database is modified by the addition of a new restaurant object, with name "Hakata," as described in Example 7.2. Therefore, at t_3, the result R_3 of the polling query contains the new restaurant object in addition to the two old restaurant objects. The new restaurant object is detected by the differencing algorithm. Accordingly, the DOEM database maintained by QSS now includes the new restaurant object, with a create annotation $cre(t_3)$ on it. Note also that at this time, *t[-1]* $= t_2$, so that this create annotation satisfies the predicate in the where clause of Q_c. Therefore the result of the query Q_c over the modified DOEM database contains the new restaurant object "Hakata," and the user is notified of this result. □

☆U.S. GOVERNMENT PRINTING OFFICE: 1999-610-130-81136

DISTRIBUTION LIST

addresses	number of copies
CRAIG S. ANKEN AFRL/IFTB 525 BROOKS ROAD ROME NY 13441-4505	5
PROF. JENNIFER WIDOM COMPUTER SCIENCE DEPT. STANFORD UNIVERSITY PALO ALTO CA 94305	1
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

1

COMMANDER, CODE 4TL000D
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

1

COR, US ARMY AVIATION & MISSILE CMD
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

2

REPORT LIBRARY
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

1

ATTN: D'BORAH HART
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

1

AFIWC/MSY
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

1

ATTN: KAROLA M. YOURISON
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

1

USAF/AIR FORCE RESEARCH LABORATORY
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

1

ATTN: EILEEN LADUKE/D460
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

1

OUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202 — 1

SOFTWARE ENGR'G INST TECH LIBRARY
ATTN: MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890 — 1

USC-ISI
ATTN: DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695 — 1

KESTREL INSTITUTE
ATTN: DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304 — 1

ROCHESTER INSTITUTE OF TECHNOLOGY
ATTN: PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700 — 1

AFIT/ENG
ATTN:TOM HARTRUM
WPAFB OH 45433-6583 — 1

THE MITRE CORPORATION
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730 — 1

UNIV OF ILLINOIS, URBANA-CHAMPAIGN
ATTN: ANDREW CHIEN
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801 — 1

HONEYWELL, INC.
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102 — 1

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA 1
ATTN: DR. YIGAL ARENS
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE 1
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

AFIT/ENG 1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

NSA/OFC OF RESEARCH 1
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

AT&T BELL LABORATORIES 1
ATTN: MR PETER G. SELFRIDGE
ROOM 3C-441
600 MOUNTAIN AVE
MURRAY HILL NJ 07974

ODYSSEY RESEARCH ASSOCIATES, INC. 1
ATTN: MS MAUREEN STILLMAN
301A HARRIS B. DATES DRIVE
ITHACA NY 14850-1313

TEXAS INSTRUMENTS INCORPORATED 1
ATTN: DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265

KESTREL DEVELOPMENT CORPORATION 1
ATTN: DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

DARPA/ITO
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714 1

NASA/JOHNSON SPACE CENTER
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058 1

STERLING IMD INC.
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440 1

HUGHES SPACE & COMMUNICATIONS
ATTN: GERRY BARKSDALE
P. O. BOX 92919
BLDG R11 MS M352
LOS ANGELES, CA 90009-2919 1

SCHLUMBERGER LABORATORY FOR
COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720 1

DECISION SYSTEMS DEPARTMENT
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421 1

SOUTHWEST RESEARCH INSTITUTE
ATTN: BRUCE REYNOLDS
6220 CULEBRA ROAD
SAN ANTONIO, TX 78228-0510 1

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899 1

EXPERT SYSTEMS LABORATORY
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604 1

NAVAL TRAINING SYSTEMS CENTER 1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

DR JOHN SALASIN 1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR BARRY BOEHM 1
DIR, USC CENTER FOR SW ENGINEERING
COMPUTER SCIENCE DEPT
UNIV OF SOUTHERN CALIFORNIA
LOS ANGELES CA 90089-0781

DR STEVE CROSS 1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

DR MARK MAYBURY 1
MITRE CORPORATION
ADVANCED INFO SYS TECH; G041
BURLINTON ROAD, M/S K-329
BEDFORD MA 01730

ISX 1
ATTN: MR. SCOTT FOUSE
4353 PARK TERRACE DRIVE
WESTLAKE VILLAGE,CA 91361

MR GARY EDWARDS 1
ISX
433 PARK TERRACE DRIVE
WESTLAKE VILLAGE CA 91361

DR ED WALKER 1
BBN SYSTEMS & TECH CORPORATION
10 MOULTON STREET
CAMBRIDGE MA 02238

LEE ERMAN 1
CIMFLEX TEKNOWLEDGE
1810 EMBACADERO ROAD
P.O. BOX 10119
PALO ALTO CA 94303

DR. DAVE GUNNING 1
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DAN WELD 1
UNIVERSITY OF WASHINGTON
DEPART OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

STEPHEN SODERLAND 1
UNIVERSITY OF WASHINGTON
DEPT OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

DR. MICHAEL PITTARELLI 1
COMPUTER SCIENCE DEPART
SUNY INST OF TECH AT UTICA/ROME
P.O. BOX 3050
UTICA, NY 13504-3050

CAPRARO TECHNOLOGIES, INC 1
ATTN: GERARD CAPRARO
311 TURNER ST.
UTICA, NY 13501

USC/ISI 1
ATTN: BOB MCGREGOR
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292

SRI INTERNATIONAL 1
ATTN: ENRIQUE RUSPINI
333 RAVENSWOOD AVE
MENLO PARK, CA 94025

DARTMOUTH COLLEGE 1
ATTN: DANIELA RUS
DEPT OF COMPUTER SCIENCE
11 ROPE FERRY ROAD
HANOVER, NH 03755-3510

UNIVERSITY OF FLORIDA 1
ATTN: ERIC HANSON
CISE DEPT 456 CSE
GAINESVILLE, FL 32611-6120

CARNEGIE MELLON UNIVERSITY 1
ATTN: TOM MITCHELL
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890

CARNEGIE MELLON UNIVERSITY 1
ATTN: MARK CRAVEN
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890

UNIVERSITY OF ROCHESTER 1
ATTN: JAMES ALLEN
DEPARTMENT OF COMPUTER SCIENCE
ROCHESTER, NY 14627

TEXTWISE, LLC 1
ATTN: LIZ LIDDY
2-121 CENTER FOR SCIENCE & TECH
SYRACUSE, NY 13244

WRIGHT STATE UNIVERSITY 1
ATTN: DR. BRUCE BERRA
DEPART OF COMPUTER SCIENCE & ENGIN
DAYTON, OHIO 45435-0001

UNIVERSITY OF FLORIDA 1
ATTN: SHARMA CHAKRAVARTHY
COMPUTER & INFOR SCIENCE DEPART
GAINESVILLE, FL 32622-6125

KESTREL INSTITUTE 1
ATTN: DAVID ESPINOSA
3260 HILLVIEW AVENUE
PALO ALTO, CA 94304

USC/INFORMATION SCIENCE INSTITUTE 1
ATTN: DR. CARL KESSELMAN
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

MASSACHUSETTS INSTITUTE OF TECH 1
ATTN: DR. MICHAELE SIEGEL
SLOAN SCHOOL
77 MASSACHUSETTS AVENUE
CAMBRIDGE, MA 02139

USC/INFORMATION SCIENCE INSTITUTE 1
ATTN: DR. WILLIAM SWARTHOUT
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

STANFORD UNIVERSITY 1
ATTN: DR. GIO WIEDERHOLD
857 SIERRA STREET
STANFORD
SANTA CLARA COUNTY, CA 94305-4125

NCCOSC RDTE DIV D44208 1
ATTN: LEAH WONG
53245 PATTERSON ROAD
SAN DIEGO, CA 92152-7151

SPAWAR SYSTEM CENTER 1
ATTN: LES ANDERSON
271 CATALINA BLVD, CODE 413
SAN DIEGO CA 92151

GEORGE MASON UNIVERSITY 1
ATTN: SUSHIL JAJODIA
ISSE DEPT
FAIRFAX, VA 22030-4444

DIRNSA 1
ATTN: MICHAEL R. WARE
DOD, NSA/CSS (R23)
FT. GEORGE G. MEADE MD 20755-6000

DR. JIM RICHARDSON 1
3660 TECHNOLOGY DRIVE
MINNEAPOLIS, MN 55418

LOUISIANA STATE UNIVERSITY 1
COMPUTER SCIENCE DEPT
ATTN: DR. PETER CHEN
257 COATES HALL
BATON ROUGE, LA 70803

INSTITUTE OF TECH DEPT OF COMP SCI 1
ATTN: DR. JAIDEEP SRIVASTAVA
4-192 EE/CS
200 UNION ST SE
MINNEAPOLIS, MN 55455

GTE/BBN 1
ATTN: MAURICE M. MCNEIL
9655 GRANITE RIDGE DRIVE
SUITE 245
SAN DIEGO, CA 92123

UNIVERSITY OF FLORIDA 1
ATTN: DR. SHARMA CHAKRAVARTHY
E470 CSE BUILDING
GAINESVILLE, FL 32611-6125

AFRL/IFT 1
525 BROOKS ROAD
ROME, NY 13441-4505

AFRL/IFTM 1
525 BROOKS ROAD
ROME, NY 13441-4505

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.